Microsoft DevOps Solutions: Planning Deployment Environment Strategies

UNDERSTANDING RELEASE STRATEGIES

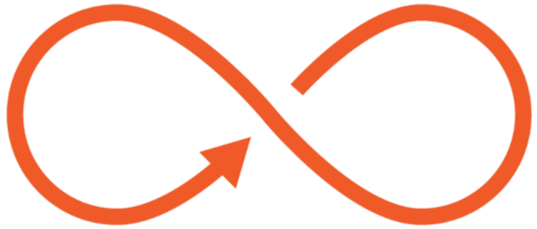
Chris B. Behrens
SOFTWARE ARCHITECT
@chrisbbehrens

A Different Kind of Course

Narrower focus and shallower depth

Deployment is my jam – check out my other courses

Release Strategies on the Test



A finite number of patterns

The exam's coverage is good for the real world

These are patterns you'll need in real life

The Pattern List

Blue – Green Deployments

Rolling Deployments

Ring or Canary

Blue-Green Deployment

Two separate production-possible environments

“Production” is a designation, not a dedicated set of resources

The designation is a network designation

You have a Staging/Test/Verification environment

You’re almost there already

Without Blue-Green, you push Staging code to production, but with production transforms

The Other Fifth of the Way to Blue-Green

Flip your network to point to Staging

What WAS production becomes staging, and vice versa

This buys you rollback…

Sort of

Network Considerations for Deployment

“simply flip your network”

It’s 10 AM in the morning somewhere for your audience

Draining

Gets your users from the old server to the new one

Azure Application Gateway

The old server stops getting new connections

This is a widely used pattern

We want a hot backup in case our deployment didn’t go as planned, and we want to make rollback as simple as possible

Rolling Deployments

More than one server

Deployment on a gradual basis

Backward Compatibility

If you think about it for a minute

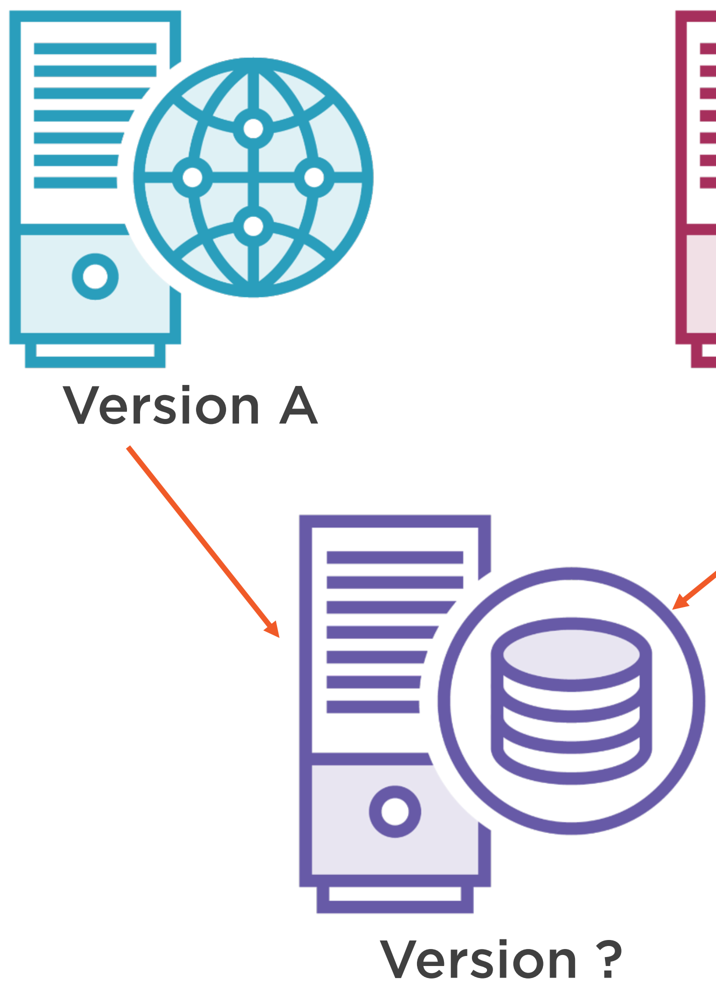
Backward compatible with the previous version of the code

Otherwise, the mixed environment will break

To remove a function, remove the possibility of request first

And then remove the function on the next deployment

The Key Concept in Rolling Deployments



Risk Mitigation for Rolling Deployments

Introduces risk gradually

So only 10% of your servers can be in trouble

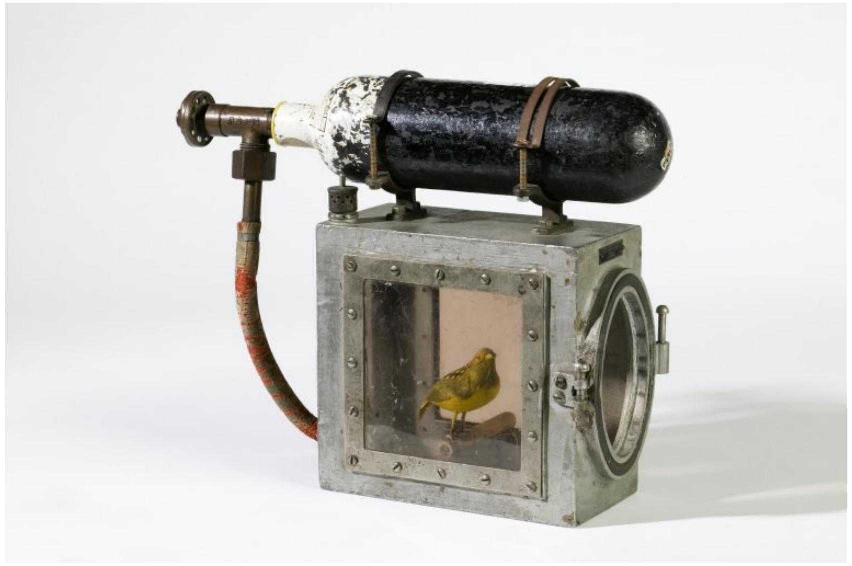
You obviously need to have ten servers to be able to deploy to 10% of them

Canary / Ring Deployments

A more sophisticated version of rolling deployments

The difference in versions is the point

What’s the Canary?



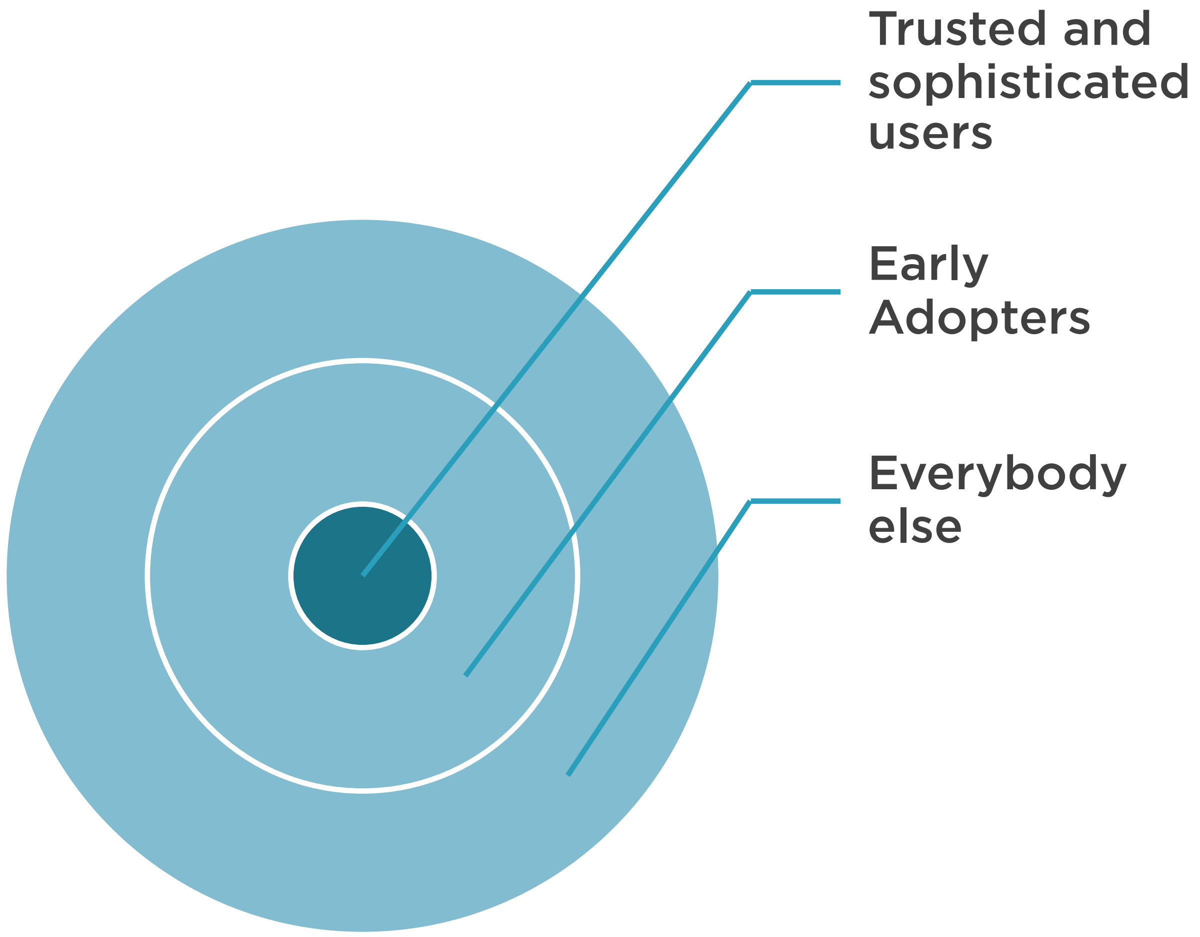
Canaries were kept in mines

Canaries’ systems were very sensitive to CO and other toxic gases

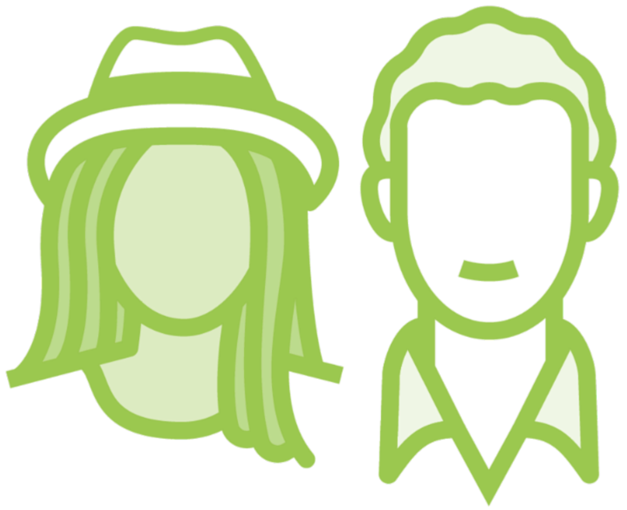
When the canaries fell over, it was time to get everybody out of the mine

Miners took good care of their canaries, and so should you

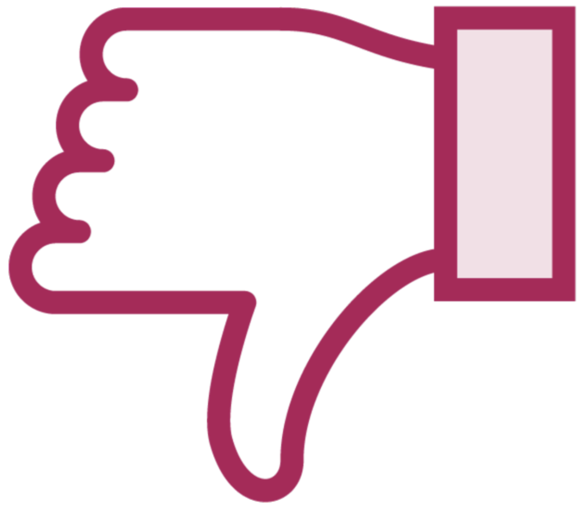
Who's the Canary?



Risk Mitigation for Canary Deployments



Find yourself some trusted users



And find out what they think of your new release

(Not Much) Azure Tooling for All of This

Draining is supported in the interface

There's no "Roll Deployment" button

You'll mostly have to roll Canary deployments yourself

With one minor exception

Blue Green Tooling with Deployment Slots

Deployment slots enable the Blue-Green pattern

But they can only be a part of the solution

Taking into account the rest of your infrastructure

The Essential Problem of Database Availability

1. Reconcile the existing state with the changes
2. Blow away what’s there and replace it with the changes
3. Create a brand-new environment and replace the old one

The Problem with Databases

Databases cannot be recreated from nothing (ex nihilo) every time

To do that we would need:

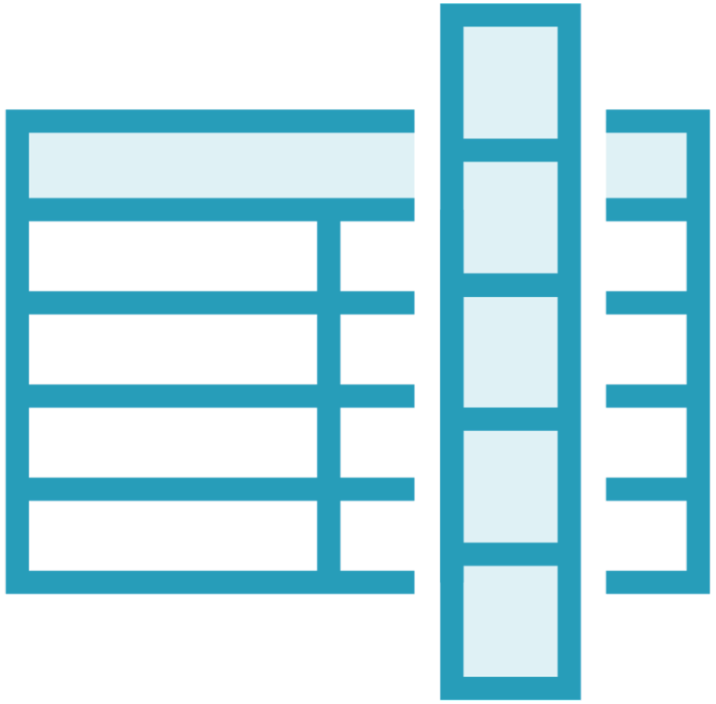
- The schema
- The data

Does your data belong in version control? Maybe not

Getting sensitive stuff out of version control is a huge pain

And if your database isn't read-only, this is all academic

What’s Happening Here



Read-only databases are rarely useful

Your customers are constantly rewriting the code necessary to regenerate your database

Database availability is the fixed point you’ve got to work around

Backward compatibility helps, but…

It doesn’t maintain rollback-ability

Some schema changes are truly irreversible, like a column drop

Solutions to This

“No irreversible schema changes”

Better get a good layer of abstraction over that

Transitional deployment to bridge the gap

Code which can run on EITHER version

Key to the version

Schema-on-Write

Transactional database are schema-on-write

Schema enforced at INSERT or UPDATE

This works well for transactional considerations

But it binds a version of the code tightly to the schema

Schema-on-Read

The database ruins all our fun

Unless we look at a more document-oriented store for our application

Pay close attention to the database for your deployments

Some of my other courses on this topic:
https://app.pluralsight.com/library/courses/deploying-databases-octopus

https://app.pluralsight.com/library/courses/microsoft-azure-web-applications-services-deploying

How Many Nines?

Percentage Uptime	Downtime Per Year
99%	Four days
99.9%	Nine hours
99.99%	One hour
99.999%	Five minutes

Our Deployment Strategies

1. ~~Reconcile the existing state with the changes~~
2. ~~Blow away what’s there and replace it with the changes~~
3. Create a brand-new environment and replace the old one

Two Patterns of Deployment

Snowflake server – a server with a unique configuration

Deploying means reconciling the old state to the new

Immutable Server

Every deployment is the creation, ex nihilo, of an entirely new server

A server whose state cannot change

A server whose state is unrelated to a previous server

Because of that, we can unit, integrate, load, and security test it at our leisure

“With no human intervention”

There was MASSIVE human intervention, just earlier in the process